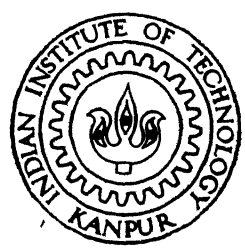
An Interpreter for a Message Centric Software Specification System

by

S.R.S. SRINIVASU DHULIPALA



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1997

An Interpreter for a Message Centric Software Specification System

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

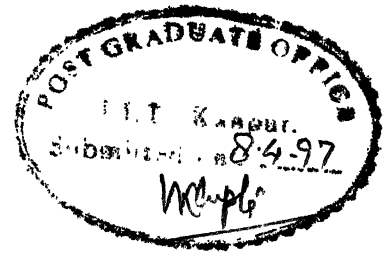
S.R.S. Srinivasu Dhulipala

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April 1997



CERTIFICATE

This is to certify that the work contained in the thesis entitled "An Interpreter for a Message Centric Software Specification system" by "S.R.S.Srinivasu Dhulipala", has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. Harish Karnick
Computer Science & Engineering Department
Indian Institute of Technology, Kanpur.

April, 1997

Acknowledgements

I am grateful to my thesis supervisor, Dr. H. Karnick, for offering me this thesis. I am indebted to the great *csealpha3* and the cute *godavari*. Especially, *godavari* for spending so many sleepless nights with me.

I cannot forget the days I spent with *raghu, jaya, vatsa, kali, kommu, praveen, tvrk, lade, chenna* and others. Finally, I would like to thank *mtech95* batch, who made my stay here memorable.

Abstract

We propose an interpreter for executing the specifications of A Message Centric Specification system. The interpreter interprets and runs the specifications. This Message Centric Specification system provides a formal language for specifying the design of an application . In this method, the system is viewed as a collection of concurrently executing objects transacting asynchronous messages between them. The interpreter handles the concurrent execution of objects. It schedules the processor between these objects. Also, it handles the passing of asynchronous messages between these objects. It parses the specifications and constructs the symbol tables. It also provides persistence for the objects. This tool helps in incremental building of the application. A persistence manager which provides persistence for objects has been developed as a part of this tool.

Contents

Chapter 1

Introduction

This chapter gives a brief introduction to the Object Oriented (OO henceforth) Methodologies, the motivation of this work and describes the organization of this thesis.

1.1 Introduction

Information systems are being developed by using numerous methodologies, techniques and tools. These methodologies and tools have been developed since the beginning of the computer era to support the ever increasing demand for computer based information systems. A good methodology and proper tools are essential to develop a large and complex system successfully. A recent trend of viewing systems as collections of identifiable, interacting objects became an important paradigm in information systems development.

Broadly, there are two kinds of methodologies - namely Structured Design methodologies and Object Oriented methodologies. Several OO Methodologies are being used for software development these days. . Some of the well known methodologies in the OO paradigm are by Booch, Rumbaugh's OMT, Jacobson's OO Software Engineering. Software developed by the OO Methodology is found to be extendible, reusable and easily maintainable.

In addition to following a methodology to develop a system, the usage of tools

that allow rapid prototyping of the system enhances the productivity and quality and allow incremental building and testing of the software.

A new Message centric OO method is described in A Message Based Specification For Object Oriented Software Construction [Sar96]. In this method, a formal language has been developed for specifying a software system(the analysis/design specification). In this thesis an interpreter that takes such a specification of a system and executes them is developed. This tool helps in incremental building of a software system.

Classes, Objects and messages/methods are central to the development of object oriented software. A class is a definition of a concept with all of its data items and the operations on that data encapsulated with in it. A class generates objects (specific instances) that contain the data specific to that instance of a class and these objects respond to messages or carry out the operations defined in the class. An operation on data defined within a class is referred to as a method. An object can invoke a method in another object by sending a message to that object.

This Message centric OO method emphasizes both the static structure and dynamic structure structure of the system being modelled. A formal textual notation, which is independent of any programming language, is used to represent the analysis/design specification.

1.2 Motivation

This Message Based Specification System is a new method, that differs from the existing methods. The aim of this thesis is to develop a tool that will help rapid prototyping of a system. The specification produced in this method , mainly, contains two parts, an Object Model and a Message model. The Object Model is used to specify the static structure of the system . Classes, their attributes and relations between these classes are given in the Object Model of the specification. The behavior of the objects is not specified in the OM. It is implicitly specified in the Message Model of the specification. The Message Model gives message passing scenarios between the objects of the system, in terms of Processes and Sub-Processes.

The Message Model specifies the dynamics of the system.

In this thesis, a tool which helps in rapidly prototyping a system is proposed. This tool is an interpreter which can execute the message based specification of the application system. Using this tool, the designer of the system can know the behavior of the system in the early stages of development. He can check if the user requirements are conformed, or clarify the requirements of the intended system before large amounts of effort are spent on developing software. Thus this tool helps incremental building of the system.

1.3 Overview of the Thesis

This section briefly describes the remaining chapters and appendices. Chapter 2 gives a brief description of the concepts of the methodology and syntax of the specification language. Chapter 3 describes the design of the tool and its various components. Chapter 4 discusses the implementation details of the tool. Chapter 5 gives conclusions and possible extensions . Appendix A gives the Registration Process of DOAA Office Automation with which this tool was tested. Appendix B gives the formal notation of the methodology in BNF.

Chapter 2

Notation

In this Chapter, the concepts of the message centric specification method are introduced and a brief description of the notation is given. A detailed description of the notation and development process in this methodology is given in [Sar96]. The notation is informally specified (The formal specification is given in Appendix B). The DOAA Office Automation example is used to explain various concepts in this Chapter. The specifications of this example are given in Appendix A.

2.1 Concepts of this Methodology

In this methodology the system is viewed as a collection of objects communicating with each other. Each object has some data that constitute its state and some rules that tell when to send a message, when to receive a message and what to do when a message is received . As all these rules involve messages, the rules are specified with the message rather than with the object.

Some basic terms that are used in this method are explained below.

Object An Object is an abstract or real world entity which has state, behavior and identity, and communicates with other objects.

State The state of an object at any instant consists of its knowledge about itself and its knowledge about the environment at that instant.

The knowledge about itself is represented by the values of the attributes of that object. The knowledge about the environment is represented by the messages it transacts with other objects.

message A message forms the unit of communication between two objects. In this specification system, a message is asynchronous.

The message is defined by the sender, receiver, message identifier and the arguments for the message. Also specified along with the message are Pre conditions (Constraints on the state of the sender to send the message), Post conditions (Constraints on the state of the receiver to receive the message) and Action to be performed by the receiver on receiving that message.

Process A Process is a sequence of events that happen to realize a meaningful real-world functionality.

Processes and objects are two central concepts of this method. In the first stage (requirements specification) there will be only processes but no classes or objects. In Analysis and Design stages there will be both classes, objects and processes. And in the implementation there will be only objects but no processes.

Generic process and Generic message A generic process is a mechanism used to represent similar turn of events once and reuse it later. Similarly generic messages can be used when similar messages are being sent to different objects. Generic Processes are used capture commonalities in the system.

Sub Process Sub process is useful to handle complexity. Large processes can be divided into sub processes. Generally, the interactions of one central object with other objects to do an operation can be put into a subprocess.

2.2 Notation

2.2.1 Object Model

The object model is used to represent the static structure of the system. There are three kinds of objects in the system : entity objects, interface objects and control objects. Entity or data objects are used to store information. They usually exist in the problem domain and represent some real-world entity or concept. Interface objects are used to view data in entity objects. These are for users. Control objects are used to collaborate between several entity and interface objects. The notation for entity and control objects is same and is different from notation for interface objects.

Class Specification This specification is valid for entity and control objects.

CLASS : *Name of the class*

Name of the class is an identifier.

This declaration begins declaration of a new class. All the items that follow this declaration belong to this class. Only another 'CLASS : xxx' or end-of-file will end the declaration of this class. All the other fields that follow CLASS are optional.

TYPE : ABSTRACT or NONPERSIST

This field is optional. If nothing is specified about the *TYPE*, then it will be taken as non-abstract data class.

An ABSTRACT class is one which does not have any instance. A class can be an abstract class only if it is a base class of an inheritance hierarchy.

A NONPERSIST class is one whose objects do not need persistence.

For example, in the DOAA system, 'Person' is an abstract class whose derived classes are STUDENT, FACULTY_MEMBER, DOAA etc. A class derived from an abstract class can also be an abstract class.

INHERITS : *class name [,class name ...]*

The list of classes from which this class inherits.

KEY : *attribute name*

The name of the key attribute, the purpose of which is to allow a user to log into the system asking for the key value.

ATTR :*attr_name <type> [, attr_name <type> ...]*

list of attributes each specified as attribute_name <type>. type can be any of

- *STRING, NUMBER, DATE.. etc.* (i.e. scalar type)
- *[attr_decl]* (represents list)
- *TABLE [attributes list]* (a table)
- { *attributes_list*} (set of dissimilar but related items)

Example : Student has attribute roll_no. roll_no <NUMBER>

INTERFACE_CLASS : class_name

If an object of this class needs an interface to the outer world, the specified interface class is used.

GENERALIZATION OF : *class_name [, class_name ...]*

class_name [, class_name ...] are classes which inherit from this class.

AGGREGATION OF : *Class name (multiplicity) [, Class name (multiplicity)]*.

PART OF : *Class name (multiplicity)[, Class name (multiplicity)]*

multiplicity is the number of part of objects of the given class in the aggregate object .

RELATED WITH :

RelatedClass name (relation_name, multiplicity) [, RelatedClass name (relation_name, multiplicity)]

relation_name is the role the related object plays with respect to this object in the relation.

multiplicity specifies how many instances of the RelatedClass are related with one instance of this class.

Interface object specification *CLASS NAME* : *Name of the interface class*

TYPE : *INTERFACE*

MENU ITEMS : {*id* : 'menu_identifier',
name : 'name of the menu',
action : Action to be taken when this menu item is selected,
description : help message,
} [,{...}]

menu_identifier is used to uniquely identify this menu item. In addition to normal identifier syntax, '/' symbol can be used to denote the hierarchy of menu items.

name is the name of the menu item that is displayed to the user.

description is the message to be displayed when the user asks for help about this menu item.

Action will be one of

- send message msg_id [of process_id [to object]]
- do sub_process subproc_id
- invoke_menu menu_id
- select menu_item 'some_other_menu_item'

- select fillin_item 'a_fillin_item'

The actions are explained in Message specifications.

FILL_IN ITEMS : { id : 'fillin_identifier',
name : 'name of fill in item',
constr : Constraints on the value of this fillin item,
action : Action to be taken when this item is filled,
attached_to : is this item attached to an attribute
of some other class
}

constraints can used to define either type or value constraints on the fill_in item.

attached_to is useful to initialize the values of fill_in items when required.

DISPLAY METHOD : These are used to specify logical display of screen with fill_in and menu items.

INHERITS : *class_name [, class_name ...]*

GENERALIZATION OF : *class_name [, class_name ...]*

The meaning of inheritance for interface classes differs slightly from that for normal classes. If interface class A inherits from interface class B, then A can use or replace some or all fillin items and display methods of B, and can also extend those menu items of B whose 'actions' are invoke_menus.

RELATED WITH : *CLASS (relation, multiplicity)*

This has the same semantics as in normal class definitions.

CLASS in RELATED WITH is used to specify which classes the interface object needs to attach to its fill_in items.

2.2.2 Message model

process :: process_name

sub process:: sub_process_name Division into sub processes is done when there is a central object that is gathering data from several objects to do certain operations.

generic sub process:: generic_sub_process (parameters) inputs : parameters_varying

The declaration of a set of messages begins with one of the above three process, sub process or generic sub process declarations.

A message declaration consists of the following fields.

FROM: Class name

TO: Class name (constraint)

Class name is the name of the class specified in the object model.

constraint can be

- condition on some attribute of the object. This is used to identify a particular instance of the class.
  (variable == class::attribute) (class::attribute1 = class::attribute2)
- ALL, meaning this message has to be set to all the instances of the class.
- ALL_RELATED, meaning this message has to be sent to all those instances of receiver class that are related to this object.
- handle == ref
  handle == BACK
  The above means that we explicitly give the handle (handle identifies an object in the system. Every object will have a unique handle. In the context of a Programming language like C++, handle is same as reference. In the context of a Database, handle is the primary key) instead of fetching it from the relations of the sender object.

When the handle is BACK, the sender should not pick up the object handle from its Relations table, but should just send it, as a response to the message it received, to the corresponding object.

MSG_ID : message identifier (parameters) The syntax of message identifieris same as that of class name in Class specification. The syntax of the parameters is the same as syntax of 'list of attributes' as described in the Class specifications.

MSG_TYPE : type of the message

type of the message can be reply, shared or exclusive.

If the type is reply, the message is being sent as a reply to a message this object has previously received. The sender of the request message waits for this reply message.

shared and exclusive represent the security level of the message. If it is specified as exclusive ,then only this object can send the message to the receiver. If it is specified as shared, then other objects can send this message to the receiver provided security level in that message specification is also specified as shared. shared is useful to restrict the senders to a set of objects.

MSG_COND : constraint that does not depend on the state of the sender, to send the message.

This is used to send a message repeatedly (looping of the message) or send a message depending on some condition which is not associated with the state of the object.

Example : if (variable == value) foreach item (table_name or list_name)

RESULT : reply to this message

The syntax of 'result' is similar to that of list of attributes.

When the 'result' is specified, the sender 'expects' some response to this message. This response is the result of action taken by the receiver of the message.

PRE_COND : condition on the state of the sender before sending the message

POST_COND : condition on the state of the receiver before receiving the message

It is the combination (AND/OR) of one or more of the following :

- received msg_id [of process_id]
- received result_of msg_id [of process_id]
- sent msg_id [of process_id]
- attribute <=> value
- attribute1 <=> attribute2

The Pre condition is a constraint on the state of the sender to send the message. Post condition is a constraint on the state of the receiver to receive the message.

By specifying different Post conditions for the same message exchanged between same two objects, we can specify different actions depending on when the message is received. Other use of Post condition is to decide whether to accept or reject the message depending on its time of arrival. Pre and Post conditions are also important for testing purposes.

Action : action to be taken after this message has been received

Action can be any of the following :

- send message msg_id [of process_id]
- do sub_process sub_proc_id (parameters)
- send generic message msg_id
- do generic sub_process process_id (parameters) {values to substitute}

generic message:: This is declaration of normal message, except that some fields will be empty.

Variables in Messages To specify the source of a variable (an argument to the Message or a variable in a PRE_COND etc.), the following can be used :

- an attribute of sender : $variable.
- Sender receives it as an argument of some other message of this process : $variable!msg_id.
- Sender receives it as an argument of some other message of some other process : $variable!proc_id!msg_id .
- Sender receives as a result of the message that was sent by it in this process : $variable=msg_id!proc_id.
- If it is a loop variable (Eg. foreach loop etc.) : $nest_level~variable .
- If the sender of the message is an Interface Object, then the interface object may use some data which it has got as temporary input. This data can be represented by : %variables_list.
- Internal values of message fields can referred by prefixing the field with '$'. This can be useful in generic processes.

  Eg. $FROM, $TO, etc.

Comments specification Any text beginning with a character # upto the end-of-line is considered a comment in all specifications.

Chapter 3

Design

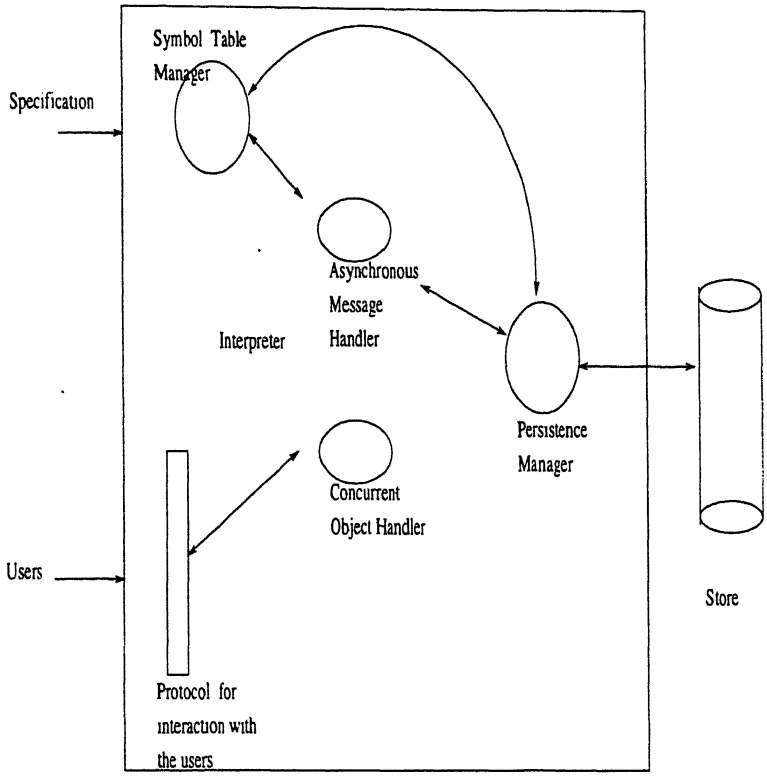
In this chapter, the design of the interpreter is described. The various components of the interpreter and their functionalities and how they cooperate with each other to execute the specifications is discussed in detail. All the examples used for illustration in this chapter are from the DOAA Office Automation application, which is given in Appendix A.

3.1 System Design

The ideal design of the tool would have been the one shown in the figure and discussed below.

The rapid prototyping of an application system in this methodology consists of two parts : the handling of asynchronous messages and the handling of the concurrent execution of objects; providing a protocol that will help in interacting with the user. The first part forms the major portion of interpreting the specifications and testing the functionality of a system, during its incremental development. The second part also needs a major effort. We concentrated on the first part, that is the handling of concurrent objects and the handling of asynchronous messages. The second part has not been implemented systematically. Only enough has been done so that the complete interpreter can run.

The whole system of the developed interpreter consists of three main components,



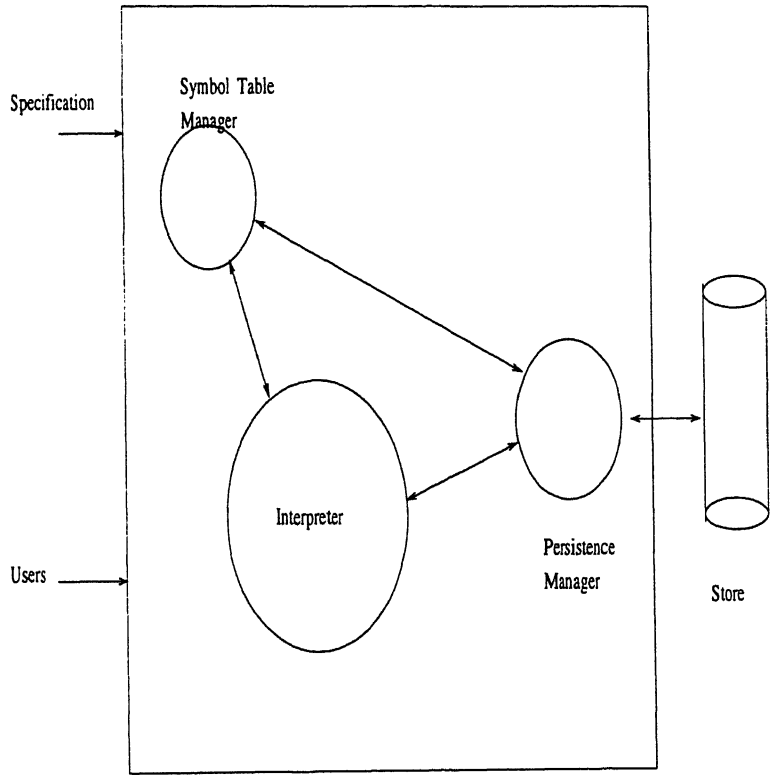
two of which provide the base for the third one-which is in fact the heart of the interpreter. They are :

1. Symbol Table Manager
2. Persistence Manager
3. Interpreter

The block diagram showing their positions in the tool and the interactions between them is given in figure. A simple description of the three components may help in understanding the block diagram. The detailed design of each of these components is discussed later in this chapter.

- Symbol Table Manager : The STM parses the specifications and maintains the various class declarations, process declarations , subprocess declarations, generic subprocess declarations and message declarations. It provides the requester with the desired declaration, when requested to do so.

- Persistence Manager : The PM provides persistence for the objects.It can be considered as the invisible layer of the tool.
- Interpreter : Interpreter is the heart of the tool. It executes the specifications of the software system, which are described in the specification language. It handles the various objects, assuming the role of each one successively, and carries out the actions of the object, and posts messages to other objects.



3.2 Symbol Table Manager

As the name itself implies, it manages the all the needed symbol tables . It basically parses the given specifications, i.e. the Object Model - the structure of the various classes and their relationships, and the Message Model- the declarations of the various processes, subprocesses, generic subprocesses and the Messages that comprise them. It acts as a repository of meta data.

The main and intended purpose of the STM is to act as a repository of all the declarations and provide them as and when needed by the other components . For example, the Persistence Manager needs the declaration of a Class while storing and retrieving objects of that class to and from the store. Also, the Interpreter needs the declarations of processes, subprocesses, generic subpr- ocesses and messages while executing the specifications.

How the STM stores various declarations is discussed below. Also the various services offered by the STM are given.

3.2.1 Class declarations

The STM stores the class declarations in a table. For fast retrieval, it uses hashing. The hashing is done on the name of the class.

An example class declaration is

```
CLASS              : PERSON
TYPE               : ABSTRACT
GENERALISATION OF  : STUDENT, FACULTY_MEMBER
ATTR               :  name <STRING>,
                      email_addr <STRING>,
                      address <STRING>
```

A class can be

- ABSTRACT, meaning this class cannot have any instances of the objects
- NONPERSISTENT, meaning the objects of this class does not need persistence

The interpreter recognizes the following basic data types of attributes

- STRING
- NUMBER
- DATE

- BOOLEAN

and complex data types

- TABLE
- LIST
- SET

The data types which are the classes declared in the object model are considered as TYPED.

STM also provides a mapping from the class names to integer ids.

3.2.2 Interface Class declarations

STM stores an interface class declaration in a table. It uses hashing which is done on the name of the interface class.

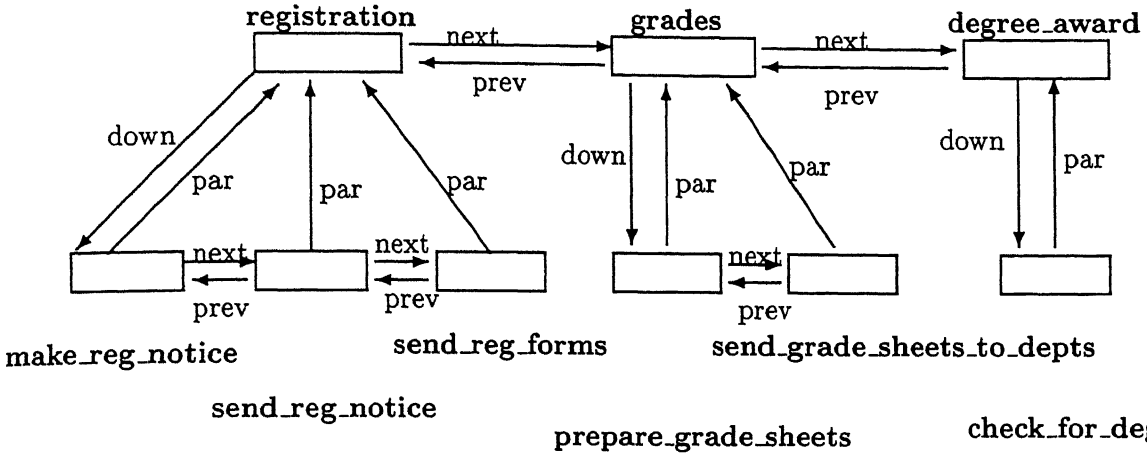
An example interface class declaration is

```
CLASS    :        DOAA_INTERFACE
TYPE     :        INTERFACE
INHERITS :  FACULTY_MEMBER_INTERFACE
MENU ITEMS : {ID : "registration"
              NAME : "Registration related"
              ACTION : INVOKE_MENU [ "registration/make_reg_notice",
                                       "registration/send_reg_notice
                                     "registration/send_reg_forms" ]
             },
             {ID : "grades"
              NAME : "Grades processing"
              ACTION : INVOKE_MENU [ "grades/prepare_grade_sheets",
                                  "grades/send_grade_sheets_to_depts"
             },
             {ID : "degree_award"
```

```
 NAME : "degree Awarding"
 ACTION : INVOKE_MENU [ "degree_award/check_for_deg_prereq'
},
{ ID : "registration/make_reg_notice"
  NAME : "Prepare Registration Notice"
  ACTION : SEND MESSAGE prepare_reg_notice OF Registration
},
{ ID : "registration/send_reg_notice"
  NAME : "Send Registration Notice"
  ACTION : SEND MESSAGE send_reg_notice_to_depts OF
                                        Registration
},
{ ID : "registration/send_reg_forms"
  NAME : "Send Registration forms to DPGCs, DUGCs"
  ACTION : SEND MESSAGE send_reg_forms_to_cdgcs OF
                                  Registration
},
{ ID : "grades/prepare_grade_sheets"
  NAME : "Prepare Student's Grade Sheets"
  ACTION : SEND MESSAGE prepare_grade_sheets OF
                                      Evaluation_process
},
{ ID : "grades/send_grade_sheets_to_depts"
  NAME : "Send Student's Grade Sheets to their Depts"
  ACTION : SEND MESSAGE students_grade_sheets OF
                        Evaluation_process
}
```

The various menu items are organized in hierarchical (tree) fashion. This is done for easy traversal.

For example, the menu items of DOAA_INTERFACE class are stored as shown in figure.



The 'par' and 'prev' pointers help the easy traversal of the menu hierarchy. While displaying the menu to a user, the interface object needs to remember the current menu item the user has selected. When the user wants to go up one level, it can be done by following the 'par' pointer. After that, the 'prev' is followed until there is no 'prev' menu. Thus the entire menu at the next higher level can be displayed.

3.2.3 Process Subprocess and Generic Subprocess declarations

STM uses separate tables to store process, sub process and generic subprocess declarations. It uses hashing which is done on the name of the process, subprocess, or generic subprocess.

An example subprocess declaration from the Registration process is given below.

```
SUB_PROCESS :: check_eligibility_for_reg
#-------------------------------------
FROM       : CONVENER
TO         : SELF
MSG_ID     : check_for_reg_notice ()
RESULT     : present <BOOLEAN>
```

```
IF ($present=check_for_reg_notice == TRUE) {

      FROM       : CONVENER
      TO         : SELF
      MSG_ID     : check_elig ()
      RESULT     : eligibile <BOOLEAN>

      IF ($eligible=check_elig == TRUE) {

           FROM       : CONVENER
           TO         : SELF
           MSG_ID     : create_reg_form ()
           RESULT     : reg_form <REGESTRATION_FORM>

           FROM       : CONVENER
           TO         : STUDENT (BACK)
        MSG_ID     : replyto_req_for_reg_form ( $reg_form=
                     create_reg_form <REGESTRATION_FORM> ,remarks =
                     "Eligible for Registration" <STRING>)
           MSG_TYPE   : reply
           ACTION     : DO SUB_PROCESS store_reg_form_and_display_remark
      } ELSE {  # eligible is FALSE

           FROM       : CONVENER
           TO         : STUDENT (BACK)
           MSG_ID     : replyto_req_for_reg_form ( remarks =
                      "Not eligible for registration" <STRING>)

           MSG_TYPE   : reply
           ACTION     : DO SUB_PROCESS store_reg_form_and_display_remark
```

```
        }
    } ELSE { # Registration notice has not been received from DOAA
    # means it is not yet time for registration

        FROM        : CONVENER
        TO          : STUDENT (BACK)
        MSG_ID      : replyto_req_for_reg_form (remarks =
                                "Not Yet Time for registration" <STRING>)
        MSG_TYPE    : reply
        ACTION      : DO SUB_PROCESS store_reg_form_and_display_remarks

}
```

3.2.4 Services offered by STM

The STM offers the following services.

- Storing the various types of declarations - class, process, generic subprocess, subprocess and generic message, in the corresponding hash tables.
- Supplying the required declaration from the corresponding table.
- Giving the structure of a specific attribute of a class.
- Giving the declaration of a specific message from a specific process subprocess or generic subprocess.
- Giving the list of names of classes from which a specific class inherits, either directly or indirectly.
- Giving the list of names of all concrete (viz. non ABSTRACT) child classes that inherit from a specific class.

3.3 Persistence Manager

The Persistence Manager (PM) provides and manages persistence of the objects. Objects and Messages are central theme of this message centered methodology.

The main purpose of the PM is to store the objects onto the disk and retrieve them from the disk, as and when needed.

The PM uses separate databases for storing objects of classes that need persistence. The PM also maintains indexes on the primary attributes of objects. Below we discuss how the PM stores and retrieves objects from the store. The index maintenance and the various services offered by the PM are also described.

3.3.1 Storing objects on to the store

The PM generates a key to store for each new object. The key is an integer. The PM maintains a list of the next keys to be used for each class that needs persistence. The PM uses 'ndbm.h' library, a library available in unix to store and retrieve objects.

For storing an object on to the store, the PM packs the data in the object into a character string. It packs the data in an object in the following order :

- attribute values,
- keys of related objects
- keys of objects of which this object is an aggregation
- keys of objects of which this object is a part

The PM packs them as follows :

- while packing a STRING, it first writes the length of the string, and then the string.
- while packing a NUMBER, it stores the string equivalent of the integer . It does not need to write the length, as the length of an integer is known for a given machine.

- while packing a TABLE, it first writes the number of records in the table, followed by each record. A record is a sequence of attributes.
- while packing a SET, it first writes the length of the set, followed by each attribute of the set.

3.3.2 Retrieving objects from the store

For retrieving an object from the disk, the PM needs the class name and the key of the object. It retrieves the packed object, which is a character string from the database of the class. The PM then unpacks it.

While unpacking an object, it unpacks in the same order, as is used for packing. It unpacks in the following order :

- attribute values,
- keys of related objects
- keys of objects of which this object is an aggregation
- keys of objects of which this object is a part

While unpacking the object, the PM constructs the 'Object' structure and fills the various fields.

3.3.3 Index Maintenance

The PM maintains indexes on the primary attributes of a class. The primary attributes are STRING, DATE, CLASS and NUMBER. The PM provides persistence for these indexes.

The PM maintains for each primary attribute, the list of its various values and the keys of the objects having that value. While storing an object on to the store, the PM checks for each of the primary attributes, whether its key is in that particular value list. If the key is not in that list, the PM adds it. While doing this, the PM also checks if the key is in some other value list of this attribute, whose value has

been modified since it was stored for the last time. If it's so, the PM removes from that list. While deleting an object from the store, the PM removes its key from those value lists of its primary attributes that contain this key.

The index maintenance helps retrieve all objects of a class with a particular value for a primary attribute. All it needs to do is search the value lists of the attribute for the particular value and retreive the objects with the keys in the list. This index maintenance avoids the exhaustive retrieval of each and every object and checking if it satisfies the condition.

3.3.4 Services

The PM provides the following services.

- Storing an object or list of objects of a class from the store, given the keys.
- Retreiving an object or all objects of a class from the store.
- Retreiving all objects having a particular value for a primary attribute.

3.4 Interpreter

As said earlier, Interpreter is the core component of the tool. The other two components, STM and PM, basically provide the ground for it to build upon. The Interpreter interprets and executes the specifications of the application system.

As any software system interacts with users, the desired software system being prototyped will have a set of roles a user can assume. In this Message Centered methodology the Interface Classes provide the interfaces of the various roles a user can assume in the system. The Interpreter allows the users to login to the system using these Interface Classes. Some user roles may need special privileges, which necessitates authentication - some sort of password mechanism. That is beyond the scope of this work.

In this specification system, all the objects are considered to be executing concurrently. The messages that are transacted between the objects are asynchronous.

Since only one processor is available, the Interpreter has to schedule the objects and execute each of them. The Interpreter maintains a queue of all the objects that are currently active. For faster access, it stores them in hashed queues. The hashing is done on the class identifier and key.

The functionality of the Interpreter is described below.

3.4.1 Logging in the users

Any process in an application system is started by a user transacting with the system in one of the various ways that are allowed to him. This process is carried out by the various objects in the system communicating with each other.

To interpret the given specifications, the Interpreter has to allow users to login to the system. The Interpreter gets the list of all the roles of users from the STM. This is done by getting the list of all the concrete child classes, that inherit from the class 'PERSON'. These are the only roles a user can assume in the application system.

For example, in the DOAA Office Automation system, the classes DOAA, DUGC, DPGC, FACULTY_MEMBER, UG_STUDENT, MTECH_STUDENT, PHD_STUDENT and COURSE_COORDINATOR are the concrete classes which inherit from the class PERSON. These are the only roles a user can assume in the DOAA Office Automation System.

The Interpreter provides the user with the list of roles, and prompts him to select one. The key attribute value is taken as input from the user. This is to ensure that such a user exists and also to associate the interface class object to that data object.

For example, in the DOAA Office Automation system, class DOAA has the key attribute as 'pf_no¡NUMBER¿'.

The Interpreter then adds the interface object as well as the data object to the incore object queue.

3.4.2 Scheduling of objects

Since only one processor is available, the Interpreter has to schedule the objects and give control to each of them. The Interpreter gives control to the objects in a Round-Robin fashion. Here, by giving control, the Interpreter is assuming the role of that object.

Each of these objects have a queue of messages waiting for them. When the Interpreter assumes the role of an object, the Interpreter checks if the object is waiting for a response. If the object is waiting for a response to a previous message it has sent and a reply has not yet arrived the Interpreter gives control to the next object. If an object is not waiting for a response or the response has arrived, the Interpreter executes one of the waiting messages. Then it gives control to the next object.

When all the users log out of the system, the Interpreter checks if any of the objects that needs persistence has been modified. If so, the Interpreter asks the PM to store the object back onto the store and removes the object from the incore-object queue. When the incore-object queue contains no objects, the interpretation of the application is over.

The high level algorithm used for the scheduling of objects is given below in pseudo C code.

```
/* here 'current object' is the object to which the Interpreter
  has currently given control.
*/
while (still objects in the queue)
{
     if (all interface objects gone)
   {
      /* all users logged out */
      if (no data object has waiting messages or
           waiting for a response )
      {
         for each (data object that needs persistence)
```

```
        {
            if (its data is modified) /* checked by means
                    a dirty bit set */
               ask PM to store back;
            delete the object;
        }
        return;
      }
    }
    if (current object has waiting messages)
    {
         execute the object;
         if (object needs to be removed from the queue)
                 remove;
         give control to next object in the queue;
         continue;
    }
    else {
         if (current object is an interface object)
         {
             display menu and user selection;
             if (user wants to log out)
                  remove the object from incore queue;
             give control to next object in the queue;
             continue;
         }
    }
}
```

3.4.3 Executing an object

While scheduling the objects in the incore queue, the Interpreter gives control to each of them and assumes the role of that object.

If the object is waiting for response to a message it has sent, the Interpreter checks for the response. That is, it checks for the reply message. If the reply message has not yet arrived, the Interpreter gives control to the next object. If the reply has arrived, the Interpreter checks if the reply message has got any action to do and executes it.

For example, in the Registration process, STUDENT sends the message 'req_for_reg_form' to CONVENER. The message is

```
FROM       : STUDENT
TO         : CONVENER ( REL )
MSG_ID     : req_for_reg_form ()
RESULT     : res_val {reg_form <REGESTRATION_FORM>, remarks <STRING>}
ACTION     : DO SUB_PROCESS check_eligibility_for_reg
```

CONVENER executes the action of this message, a subprocess, and gives the reply message back with the response (result). The reply message is,

```
FROM       : CONVENER
TO         : STUDENT (BACK)
MSG_ID     : replyto_req_for_reg_form ( $reg_form=create_reg_form
  <REGESTRATION_FORM> ,remarks = "Eligible for Registration" <STRING>)
MSG_TYPE   : reply
ACTION     : DO SUB_PROCESS store_reg_form
```

Here, the reply message has got an action to do, a subprocess.

When the object is doing a subprocess, the Interpreter gets the next message to be executed in that subprocess and executes it.

For example, in the subprocess 'process_the_reg_form', DOAA ,after executing the message 'extract_details', has to execute the next message 'check_student_for_hall_dues'.

When the object is not waiting for any response and not doing any subprocess, the Interpreter takes the first message in the waiting queue of messages and executes it. If the sender of that message needs a response and is waiting for it, the object has to send back a reply message after executing the action associated with it. The action may be to send a message or do a subprocess.

For example, in the Registration process, STUDENT sends the message 'req_for_reg_form' to CONVENER. The message is

```
FROM      : STUDENT
TO        : CONVENER ( REL )
MSG_ID    : req_for_reg_form ()
RESULT    : res_val {reg_form <REGESTRATION_FORM>, remarks <STRING>}
ACTION    : DO SUB_PROCESS check_eligibility_for_reg
```

STUDENT after sending the message waits for the response. CONVENER after executing the action of this message, which is to do subprocess 'check_eligibility_for_reg', sends back the reply message with the result.

If the control object belongs to an interface class, the Interpreter checks if any messages are waiting. It executes one of them and gives control to the next object in the incore-object queue. If no messages are waiting, the Interpreter displays the menu list to the user and takes the selection from the user. Then it will execute the action associated with that menu item . The action associated with a menu item will be one of

- send msg_id of proc_id
- invoke menuitems list of menuitems, this happens when there is a sub menu associated with this menu item.
- do subprocess proc_id

For example, in the DOAA_INTERFACE interface class, the menu item 'registration' has an action to invoke a sub menu. The menu item is given below.

```
{ ID : "registration"
  NAME : "Registration related"
  ACTION : INVOKE_MENU [ "registration/make_reg_notice",
                         "registration/send_reg_notice",
                         "registration/send_reg_forms" ]
}
```

If the action is to send a message, the Interpreter sends the message.

For example, in the DOAA_INTERFACE class, if DOAA selects the menu item 'registration/send_reg_notice', the action is to send a message. The menu item is given below.

```
{ ID : "registration/send_reg_notice"
  NAME : "Send Registration Notice"
  ACTION : SEND MESSAGE send_reg_notice_to_depts OF Registrati
}
```

3.4.4 Executing a Message

When executing a message an object has recieved, the Interpreter checks if the message has any post condition . The post condition of a message is a condition on the receiver to execute that message. If there is a post condition, the Interpreter checks it. If the post condition is not satisfied, the object has to ignore that message. If satisfied, it has to execute the action associated with it.

If a message has no action specified in the specification, user should provide code for that message. The Interpreter calls that code.

For example, in the 'process_reg_form' subprocess, DOAA sends a message 'check_student_fo to ACADEMIC_SECTION. The message has no action associated with it. The message is given below.

```
FROM      : DOAA
TO        : ACCOUNTS_SECTION
MSG_ID    : check_student_for_instt_dues ($roll_no=extract_detail
```

```
                                        !process_the_reg_form <NUMBER>)
        RESULT      : payed<BOOLEAN>
```

If a message has some action specified for it in the specifications, the Interpreter executes that action. The action may be one of the following :

- send msg_id of proc_id
- do subprocess proc_id or do generic subprocess proc_id

Send a Message

If the action is of type 'send msg_id of proc_id', the Interpreter gets the declaration of the message which has to be sent. It then checks if there is any pre condition defined for that message. If there is a pre condition, to send that message the sender has to satisfy that condition. If that pre condition is not satisfied, the sender will not send the message.

For example, in the Registration process, DOAA_INTERFACE has to send a message 'send_reg_notice_to_depts' to related DOAA object, when DOAA selected the menu item 'registration/send_reg_notice'. The message is given below.

```
FROM        : DOAA_INTERFACE
TO          : DOAA (REL)
MSG_ID      : send_reg_notice_to_depts ($notice=prepare_reg_notice
                                                !Registration<NOTICE>)
PRE_COND    : RECEIVED RESULTOF prepare_reg_notice
ACTION      : SEND MESSAGE notice_for_registration
```

Here to send this message, DOAA_INTERFACE has to satisfy the pre condition that he should have recieved the result of the message 'prepare_reg_notice'.

To send the message, the Interpreter has to know the identities of receiver objects. For this, it has to evaluate the constraint on the receiver class. The various types of constraints are described in Chapter 2.

For example, in the Registration process, DOAA sends a message 'notice_for_registration' to all the CONVENERS. The message declaration is given below.

```
FROM       : DOAA
TO         : CONVENER (ALL)
MSG_ID     : notice_for_registration ($notice!send_reg_notice_to_dept
                                                    !Registration<NOTICE>)
ACTION     : DO SUB_PROCESS store_notice
```

Here the message is to send to all the CONVENERS. Since CONVENER is an abstract class, the message is to be sent to all the objects of its concrete child classes, DUGC and DPGC. The Interpreter gets the keys of all those objects from the PM and posts the message to them.

The message to be sent may have some parameters. The interpreter has to get them. A parameter's source can be one of :

- attribute of sender,
- a parameter of a message it has received,
- a result of a message it has sent,
- a for loop variable.

For example, in the Registration process, DOAA sends a message 'notice_for_registration' to all the CONVENERS. The message is

```
FROM       : DOAA
TO         : CONVENER (ALL)
MSG_ID     : notice_for_registration ($notice!send_reg_notice_to_dept
                                                    !Registration<NOTICE>)
ACTION     : DO SUB_PROCESS store_notice
```

Here, the parameter is the parameter DOAA has received with the message 'send_reg_notice_to_depts'.

After getting the receiver ids and the parameters, the Interpreter posts this message to the receivers. For this, it adds this message to the waiting list of messages of the receivers. If the sender needs response, the Interpreter marks its state as awaiting response, else it moves the sent message to the done list of messages for the sender.

Do a Subprocess

When the action of a message is to do a subprocess, the Interpreter checks if the subprocess has been defined. If not defined, user has to provide code of the subprocess.

If the subprocess has been defined, the Interpreter gets the subprocess and the first message in that subprocess and executes it.

For example, in the Registration process, the action of the message 'req_for_reg_form' is to do a subprocess. The message is given below.

```
FROM      : STUDENT
TO        : CONVENER ( REL )
MSG_ID    : req_for_reg_form ()
RESULT    : res_val {reg_form <REGESTRATION_FORM>, remarks <STRING>}
ACTION    : DO SUB_PROCESS check_eligibility_for_reg
```

Here, CONVENER after receiving the message executes the first message of the subprocess, 'check_eligibility_for_reg'.

3.4.5 Displaying menu

If the object is an interface object, the user is provided with the various menu items, that have been defined in the interface class.

For example, DOAA has an associated interface class DOAA_INTERFACE, whose declaration was given earlier in this chapter.

As described earlier, the menu items are organized in an hierarchical fashion. The user can choose to exit, in which case the interface object is removed from the incore-object queue.

The selected menu item can have one of the actions, as specified in Chapter 2.

- If the action is to send a message, the Interpreter sends the message.
- If the action is to do a subprocess, the Interpreter starts the subprocess and executes the first message in that subprocess.

- If the action is to invoke menu, the sub menu is invoked.

The user is also provided with an option to move one level up, in the menu hierarchy.

3.4.6 Extensions made to the original specifications

- A new field, 'INTERFACE_CLASS : interface_class_name', is added to the class declaration. If the class is a role the user can assume in the system, it's necessary to specify the interface class associated with it.

- A new field, 'KEY : key_attribute_name' is added to the class declaration. This is necessary if the class needs persistence and is one of the roles a user can assume.

- A new option 'TYPE : NONPERSISTENT' is added to the class declaration. This means the instances of the class does not persistence. By default, all objects are considered persistent.

Chapter 4

Implementation

In this chapter, the implementation details of the various components of the tool that has been proposed is described. Some of the important data structures are given. All the examples used for illustration are from the Registration process of the DOAA Office Automation application, which is specified in Appendix A.

The interpreter has been developed in C++.

4.1 Symbol Table Manager (STM)

STM uses a structure 'Clas' to store a class declaration that is parsed. The structure Clas is given below. The various fields are explained with comments.

```
struct Clas {
        char *name;          /*  name of the class */
        int  type;          /*   type of class    */
        List *inherits_from; /* list of the names of the classes
                                this class inherits from */
        List *gen_of;         /* list of the names of child classes
                                this class */
        char *key;          /* key attribute */
        char *interface; /* name of the interface class if needed *
        Attrlist *attr;         /*    list of attributes      */
```

```
        aggr_partlist *aggr_of; /* list of the the class names this
                class is an aggregation of and the multiplicities *
        aggr_partlist *part_of;/* list of the the class names this
                class is an aggregation of and the multiplicities *
        Relation *rel_with;         /*      list of relations
                                     *      the class does have
                                     */
        Clas *nextp; /* pointer to the next class declaration in the
                                hash table */
} ;
```

STM uses a structure 'InterfaceCls' for storing an interface class declaration. The structure 'InterfaceCls' is given below. The various fields are explained with comments.

```
struct InterfaceCls {
        char *name; /* name of the i/f class */
        int type; /* type of the i/f class */
        List *inherits_from; /* list of the i/f class names from
                        which this i/f class inherits */
        List *gen_of; /* list of the i/f class names that inherit
                        from this class */
        Relation *rel_with; /* list of the relations */
        MenuItem *menu;   /*      list of menu items stored
                           *      in a heirarchical fashion
                           */
        FillinItem *fillin; /*  list of the fillin items   */
        Display *displ;         /*    display methods     */
        InterfaceCls *nextp; /* pointer to the next i/f class in th
                                hash table */
};
```

The structure 'MenuItem' is for storing the menu item of an interface class. The structure is declared as below

```
struct MenuItem {
        char *id; /* id of the menu item */
        char *name; /* name of the menu item, that is provided
                       to the user */
        MenuAction *action; /* action to be taken when this menu
                       item has been selected */
        char *descript; /* help on this item */

        /*          pointers to the  menu items in the hierarchy.
         *          'par' parent ptr is used for easy traversal
         *          of the heirarchy.
         */
        MenuItem *next, *prev, *down, *par;
};
```

STM uses a structure 'Process' for storing a process as well as a subprocess declaration. The structure 'Process' is given below.

```
struct Process {
        char *pid;          /*   name of the process or subprocess *.
        MsgSeq *msg;       /*       sequence of the messages
                            *       in the process or subprocess
                            */
        Process *nextp; /* pointer to the next process or subproces
                           in the process or subprocess table */
};
```

'Msg' is a structure for holding the declaration of a message. The structure is given below.

```
struct Msg {
        char *from;         /*   msg from          */
        char *to;          /*       msg to         */
```

```
        Constraint *to_constr;  /*          constraint on the
                                 *          to class
                                 */
        char *msg_id;    /*          id of the message      */
        Param *param;  /*       parameters to the message    */
        int type;      /*       type of the message :
                        *       REPLY, SHARED, EXCLUSIVE      */
        Constraint *mcond;  /*  constraint on the message */
        Condition *pre_cond;  /*   pre-condition    */
        Condition *post_cond;   /*    post-condition    */
        Result          *result;        /*  result   */
        Action  *action;          /*   Action    */
        Msg *nextp; /* pointer to the next message in the sequence
};
```

STM uses a structure 'GenProcess' for storing a generic subprocess declaration. The structure 'GenProcess' is given below.

```
struct GenProcess {
        char *pid;      /* name of the generic subprocess      */
        List *inputs;  /*   inputs to the generic subprocess*/
        MsgSeq *msg;   /*          sequence of the messages
                        *          in the generic subprocess
                        */
        GenProcess *nextp; /* pointer to the next generic process
                                    in the hash table */
};
```

'inputs' are the inputs to the generic process. For example, the class name of the object which invoked this subprocess.

4.2 Persistence Manager (PM)

The PM uses separate databases for each of the classes that need persistence for its objects. The database for a class is created with its name. The PM uses 'ndbm.h', a library available in unix , to store and retrieve the packed objects which are textual strings. The test data has to be created before interpreting the messages.

The PM generates integer keys for the objects while storing. It maintains a list of the next keys to be used for each of these classes.

The PM uses a structure 'Object' to hold the data of an object. The structure is given below.

```
struct Object {
        int key;        /* key of the object */
        int clsid;      /* class id to which it belongs */
        ObjAttrib *attr;    /* values of the attributes
                               of the object */
        ObjRelation *rel;   /* keys of related objects */
        ObjAggrPart *aggrlist;  /* keys of objects of which this
                                   object is an aggregation of */
        ObjAggrPart *partlist;  /* keys of objects of which this
                                   object is part of */
        Object *next;
};
```

4.3 Interpreter

The Interpreter recognizes the following basic data types and implements them as follows.

- STRING - a string of characters,
- NUMBER - an integer,
- DATE - an integer in mmddyy,

- BOOLEAN - an integer, with value 1 for true and value 0 for false.

The Interpreter uses the following structure for an incore object.

```
struct IncoreObject {
        ObjId *id; /* identifier - class id and key */
        int dirty;      /* dirty bit */
        Data *data; /* data of the object */
        Mesg_SubProc *doneMsgs;/* list of done messages */
        Mesg_SubProc *waitingMsgs; /* list of waiting messages */
        Mesg_SubProc *current_msg; /*current message being handled*
        Mesg_SubProc *current_subproc;/* stack of subprocesses this
                                this object is currently doing *
        int state;/* state of the object - RESP_AWAITING or not ? *
        IncoreObject *next;/* pointer to next object in the
                                total queue */
        IncoreObject *prev; /* pointer to prev object in the
                                total queue */
        IncoreObject *nexthash; /* pointer to next object in the
                            hash queue */
        IncoreObject *prevhash; /* pointer to prev object in the
                            hash queue */
        IncoreObject *interfaceObj; /* related interface object,
                                        if needed */
        IncoreObject *dataObj; /* related data object, if needed *
};
```

Chapter 5

Conclusion

In previous chapters, the development of an interpreter, to interpret the specifications of a message centric specification system was described. This tool helps the designer of a software system to develop it incrementally. The tool handles the concurrent execution of objects and the passing of asynchronous messages between them. The interpreter first parses the specifications given and maintains all the symbol tables. It also provides persistence for objects. The main functionality of the tool is to interpret the specifications by handling the concurrent execution of objects and the passing of asynchronous messages. This tool helps the designer in rapidly prototyping an application system. A persistent manager which provides persistence to the objects has been developed as a part of the tool, using 'ndbm.h', a database library available in unix.

A tool that generates test cases automatically from the specifications has been developed simultaneously [Shr97]. The specification system is described in [Sar96]. Translation of the specifications of a software system in to an implementation in RDBMS is described in [Vij96]. Translation of the specifications into an implementation in C++ is described in [Raf96].

5.1 Limitations of the system as currently implemented

The interpreter has been run with the DOAA Office Automation application, which is given in Appendix A. The interpreter needs code for the processes and subprocesses to be part of it, as pointed out in Chapter 3. A protocol that allows the user to provide the code as a library would be helpful. This aspect is not handled in this thesis. The handling of messages and the scheduling of objects makes the specifications executable and this is the heart of the rapid prototyping tool. The use of fillin items of interface classes to get data from the user has not been implemented. The case of an object awaiting responses from more than one object has not been implemented. Persistence for messages has not been provided.

5.2 Possible extensions

1. The provision of a protocol for interacting with the user. This helps the user to provide needed code in the form of a library.

2. Persistence for messages, treating messages as objects of a certain type.

Appendix A

DOAA Office Automation Application

A.1 Description of DOAA Office

DOAA Office deals with the academic activities in the Institute. Registration of Students, Degree awarding, new Courses approval, Student's performance evaluation each semester,etc.Here Registration process is represented in detail.

Registration involves clearing the Institute dues, Hall dues, Academic registration and final registration. Academic registration for the next semester is done before the end of the current semester. Dues clearing and final registration are done only during specified dates, at the beginning of the semester.

For Academic registration, DOAA sends a notice giving the dates of registration. List of courses is displayed on the notice board. A student fills in a registration form available from the Convener, DUGC/DPGC depending on whether he/she is a UG student or a PG student. The student then selects his courses of interest from the list of courses offered(Courses may be compulsory or electives). Instructor selects some students depending on his selection procedure (for some elective courses the instructor's approval is required). After filling the registration form, student submits it to Convener, DUGC/DPGC and gets his signature.

Dues clearance is done by paying outstanding dues to the institute.

For final registration, Student submits Registration form to DOAA's office. DOAA office checks whether student has cleared dues or not, whether he is academically eligible for registration, and enters the Student's roll number on rolls. A student is registered when his roll number is entered on rolls.

A.2 Requirements Specification

A.2.1 Registration Process

DOAA sends registration notice to notice boards. DOAA sends registration forms to Conveners, DUGC, DPGC. Student requests DUGC/DPGC for registration form. Student sees notice board for list of offered courses. Student(UG) fills the registration form with the core courses and electives. For each course he selects, student requests course instructor for approval to credit that Course (if that course requires instructor's approval). Student fills the Registration form and gets DUGC/DPGC's signature. For clearing dues, the student pays the dues and gets receipt(s). Finally, the student submits the Registration form and these two receipts in the DOAA's office.

After receiving the Registration form, DOAA's office checks whether the student is academically eligible for registration or not. If he is eligible, then the student is deemed to have registered.

A.2.2 Specifications

Looking at the description of the system, we find some obvious objects like STUDENT, COURSE, DOAA, DEPARTMENT, DPGC, DUGC, etc. STUDENT will have attributes name, address, roll_no, etc. name and address are attributes for DOAA, DUGC, DPGC also. So, we can generalize these classes to form the PERSON class, whose attributes are name, address.

The complete object model is given below. In the message model, we represented a subset of the total functionality is used to test the interpreter.

#----------

```
CLASS : PERSON
TYPE : ABSTRACT
GENERALISATION OF : STUDENT, FACULTY_MEMBER
ATTR : name <STRING>,
       email_addr <STRING>,
       address <STRING>
#-----------------------------------------------------------------
CLASS : STUDENT
TYPE : ABSTRACT
INHERITS : PERSON
GENERALISATION OF : UG_STUDENT, PG_STUDENT
KEY : roll_no
ATTR : roll_no <NUMBER>,
       address <STRING>,
       joining_date   <DATE>, #date of joining the institute
       semester     < NUMBER > # Academic semester for student
RELATED WITH : COURSE (takes, N),
               PERFORMANCE_RECORD (has, 1)
#-----------------------------------------------------------------
CLASS : UG_STUDENT
INHERITS : STUDENT
INTERFACE_CLASS : STUDENT_INTERFACE
ATTR : acad_status < STRING > # regular/warning/probation  etc
RELATED WITH : DUGC (CONVENER, 1)
#-----------------------------------------------------------------
CLASS : PG_STUDENT
TYPE : ABSTRACT
INHERITS : STUDENT
GENERALISATION OF : MTECH_STUDENT, PHD_STUDENT
ATTR : leave_status <{ casual_leaves <NUMBER>, sick_leaves <NUMBER>,
                       vacation_leaves <NUMBER> }>
```

```
#-----------------------------------------------------------------
CLASS : MTECH_STUDENT
INHERITS : PG_STUDENT
INTERFACE_CLASS : STUDENT_INTERFACE
RELATED WITH : DPGC (CONVENER, 1),
               THESIS_SUPERVISOR (Th_guide, 1),
               TA_SIPERVISOR (Ta_instr, 1)
#-----------------------------------------------------------------
CLASS : PHD_STUDENT
INHERITS : PG_STUDENT
INTERFACE_CLASS : STUDENT_INTERFACE
RELATED WITH : DPGC (CONVENER, 1),
               THESIS_SUPERVISOR (Th_guide, 1),
               PHD_ACAD_DETAILS (acad_details, 1)
#-----------------------------------------------------------------
CLASS : PHD_ACAD_DETAILS
ATTR : seminars_given <TABLE [type <STRING>, title <STRING>, date <DATE>]>
#-----------------------------------------------------------------
CLASS : PERFORMANCE_RECORD
ATTR : semester <NUMBER>,
       year <NUMBER>,
       cpi< NUMBER>,
       spi< NUMBER>,
       text<STRING>,
       grades < TABLE [ c_no<NUMBER>,units<NUMBER>,grade<NUMBER>] >
#-----------------------------------------------------------------
CLASS : REG_FORM
ATTR : year< NUMBER>,
       sem< NUMBER>,
       roll_no< NUMBER>,
       room_no  < STRING >,
```

```
        student_name<STRING>,
        Thesis_superviser<STRING>,
        Financial_Asst <STRING>,
        DPGC_sign<STRING>,
        DUGC_sign  <STRING>,
        courses < TABLE [ course_no <NUMBER>, c_name <STRING>,
                              units <NUMBER>, instructor <STRING> ] >
#-------------------------------------------------------------------
CLASS : FACULTY_MEMBER
INHERITS : PERSON
KEY : pf_no
GENERALISATION OF : COURSE_COORDINATOR, CONVENER, DOAA
ATTR : pf_no <NUMBER>,
       leave_status <{ casual_leaves <NUMBER>, sick_leaves <NUMBER>,
                         vacation_leaves <NUMBER>}>
#-------------------------------------------------------------------#
CLASS : COURSE
KEY : c_no
ATTR :
        c_no <NUMBER>,
        c_name <STRING>,
        units_pg <NUMBER>,
        units_ug  <NUMBER>,
        syllabus <STRING>,
        references  <STRING>,
        approval_date  <DATE>,
        lab <NUMBER>,
        lecture <NUMBER>,
        tutorial  <NUMBER>,
        course_slots < TABLE [time_slot<NUMBER>,day<STRING>,room_no<STRING>] >
 RELATED WITH : COURSE_COORDINATOR (taught_by, N),
```

```
                STUDENT (enrolled_by, N)
#-------------------------------------------------------------------
CLASS : COURSE_COORDINATOR
INHERITS : FACULTY_MEMBER
RELATED WITH : COURSE (teaches, N)
#-------------------------------------------------------------------
CLASS : CONVENER
TYPE : ABSTRACT
INHERITS : FACULTY_MEMBER
GENERALISATION OF : DPGC, DUGC
RELATED WITH : RULE_BOOK (Rules_store, 1)
#-------------------------------------------------------------------
CLASS : DPGC
INHERITS : CONVENER
#-------------------------------------------------------------------
CLASS : DUGC
INHERITS : CONVENER
#-------------------------------------------------------------------
CLASS : DOAA
INHERITS : FACULTY_MEMBER
INTERFACE_CLASS : DOAA_INTERFACE
RELATED WITH : DPGC (DPGC, 1),
               DUGC (DUGC, 1),
               RULE_BOOK (Rules_store, 1)
#-------------------------------------------------------------------
CLASS : NOTICE_BOARD
AGGREGATION OF : NOTICE (N)
#-------------------------------------------------------------------
CLASS : NOTICE
ATTR : date <DATE>,
       issued_by <STRING>,
```

```
       title <STRING>,
       text < STRING>
#-------------------------------------------------------------------
CLASS : RULE_BOOK
RELATED WITH : RULE (has, N)
#-------------------------------------------------------------------
CLASS : RULE
ATTR : rule_id <STRING>,
       w_e_f <DATE>,
       eff_upto <DATE>,
       rule_func <STRING>
#-------------------------------------------------------------------
CLASS : TIME_TABLE
ATTR : time_table < TABLE [ c_no <NUMBER>, Instructor <STRING>, schedule
                    <TABLE [ day <STRING>, time_slot <NUMBER>] > ] >
RELATED WITH                : DEPARTMENT ( Belongs_to , 1)
#-------------------------------------------------------------------
CLASS                        : DEGREE
ATTR                         : degree_name <STRING>,
                                 description <STRING>
RELATED WITH                : DEPARTMENT (OfferedBy, M-N),
                                 CERTIFICATE (certificate, 1)
#-------------------------------------------------------------------
CLASS                        : LEAVE_FORM
GENERALISATION OF           : STDENT_LEAVE_FORM, FACULTY_LEAVE_FORM
ATTR                         :
                    from <DATE>,
                    to <DATE>,
                    type_of_leave <STRING>
#-------------------------------------------------------------------
CLASS                        : STUDENT_LEAVE_FORM
```

```
INHERITS                    : LEAVE_FORM
ATTR                        :
                    roll_no_of_stud <NUMBER>,
                    name_of_stud <STRING>,
                    stud_sign <STRING>,
                    ta_supervisor_sign <STRING>,
                    thesis_supervisor_sign <STRING>
#-------------------------------------------------------------------
CLASS                       : FACULTY_LEAVE_FORM
INHERITS                    : LEAVE_FORM
ATTR                        :
                    pf_no <NUMBER>,
                    name <STRING>,
                    sign <STRING>
#-------------------------------------------------------------------
CLASS                       : DEPARTMENT
RELATED WITH                :
         STUDENT ( Has , 1-N) ,
         FACULTY ( Has , M-N) ,
         NOTICE_BOARD ( Has , 1-1) ,
         TIME_TABLE (Has, 1-1)
#-------------------------------------------------------------------
CLASS : ACCOUNTS_SECTION
INHERITS : GENERAL_ADMINISTRATOR
#-------------------------------------------------------------------
CLASS : HALL_OFFICE
INHERITS : GENERAL_ADMINISTRATOR
#-------------------------------------------------------------------
CLASS : CALENDER
ATTR : calender <TABLE [ date <DATE>, event <STRING>,
                         semester <STRING>] >,
```

```
        current_sem <NUMBER>,
        current_year <NUMBER>
#-------------------------------------------------------------------
CLASS                                : PERSON_INTERFACE
TYPE                                 : INTERFACE
MENU ITEMS                           :
             { ID : "calender"
               NAME : "See Calender"
               ACTION : SEND MESSAGE display_calender OF queries
             },
             { ID : "notice"
               NAME : "See Notice Board"
               ACTION : SEND MESSAGE display_notices OF queries
             },
             { ID          : "student_details"
               NAME : "Show details"
               ACTION : SEND MESSAGE give_details OF queries
               DESCRIPTION : "Shows the given student's details that are not
                              restricted by the student"
             }
#-------------------------------------------------------------------
CLASS                          : STUDENT_INTERFACE
TYPE                           : INTERFACE
INHERITS                       : PERSON_INTERFACE
MENU ITEMS                     :
             { ID : "notice"
               NAME : "See Notice Board"
               ACTION : SEND MESSAGE display_notices OF queries
             },
              { ID : "register"
                NAME : "Registration"
```

```
  ACTION : INVOKE_MENU [ "register/pay_hall_dues",
                         "register/pay_instt_dues",
                         "register/get_reg_form",
                         "register/fill_reg_form",
                         "register/send_reg_form"
                       ]
},
{ ID : "leaves"
  NAME : "Leaves"
  ACTION : INVOKE_MENU [ "leaves/get_leave_form",
                         "leaves/fill_leave_form",
                         "leaves/send_leave_form"
                       ]
},
{ ID : "add_drop"
  NAME : "Add / Drop Courses"
  ACTION : INVOKE_MENU [ "add_drop/get_add_drop_form",
                         "add_drop/fill_add_drop_form",
                         "add_drop/submit_add_drop_form"]
},
{ ID : "register/pay_hall_dues"
  NAME : "pay Hall dues"
  ACTION : SEND MESSAGE pay_hall_dues OF Registration
},
{ ID : "register/pay_instt_dues"
  NAME : "pay Institute dues"
  ACTION : SEND MESSAGE pay_instt_dues OF Registration
},
{ ID : "register/get_reg_form"
  NAME : "Get Registration Form"
  ACTION : SEND MESSAGE get_reg_form OF Registration
```

```
},
{ ID : "register/fill_reg_form"
  NAME : "Fill Registration Form"
  ACTION : SEND MESSAGE fill_reg_form OF
           Registration
},
{ ID : "register/send_reg_form"
  NAME : "Submit Registration Form"
  ACTION : SEND MESSAGE submit_filled_reg_form OF
           Registration
},
{ ID : "add_drop/get_add_drop_form"
  NAME : "Get Add Drop Form"
  ACTION : SEND MESSAGE get_add_drop_form OF Add_Drop_process
},
{ ID : "add_drop/fill_add_drop_form"
  NAME : "Fill Add/Drop Form"
  ACTION : SEND MESSAGE display_method_fill_add_drop_form OF
           fill_add_drop_form_process
},
{ ID : "add_drop/submit_add_drop_form"
  NAME : "Submit Add Drop Form"
  ACTION : SEND MESSAGE send_add_drop_form OF add_drop_process
},
{ ID : "leaves/get_leave_form"
  NAME : "Get Leave Form"
  ACTION : SEND MESSAGE give_leave_form OF Leave_processing
},
{ ID : "leaves/fill_leave_form"
  NAME : "Fill the Leave Form"
  ACTION : SEND MESSAGE fill_leave_form OF Leave_processing
```

```
            }
DISPLAY METHODS :
         MENU DISPLAY : VERTICAL
RELATED WITH     : A_D_FORM_INTERFACE (uses, 1-1),
                    STUDENT (attached_to, 1-1)
#-----------------------------------------------------------------
CLASS   : REG_FORM_INTERFACE
TYPE    : INTERFACE
MENU ITEMS :
                   { ID : "store"
                     NAME : "store this reg form"
                     ACTION : SEND MESSAGE store_reg_form
                   }
FILLIN ITEMS :
                    { ID : "year"
                     NAME : "Year"
                     CONSTR : TYPE = < NUMBER>
                   },
                    { ID : "sem"
                     NAME : "Semester"
                     CONSTR :  TYPE = <NUMBER>
                   },
                    { ID : "roll_no"
                     NAME : "Roll Number"
                     CONSTR : TYPE =  < NUMBER>
                   },
                    { ID : "room_no"
                     NAME : "Room Number"
                     CONSTR : TYPE = < STRING >
                   },
                    { ID : "student_NAME"
```

```
  NAME : "Name in Block letters"
  CONSTR : TYPE =  <STRING>
},
{ ID : "fin_asst"
  NAME : "Financial_Asst"
  CONSTR : TYPE = <STRING>
},
 { ID : "th_sup"
  NAME : "Thesis_superviser"
  CONSTR : TYPE = <STRING>
},
{ ID : "convener_sign"
  NAME : "Signature OF the Convener (DPGC / DUGC)"
  CONSTR : TYPE = <STRING>
},
{ ID : "course_no"
  NAME : "Course Number"
  CONSTR : TYPE = <STRING>
},
{ ID : "c_NAME"
  NAME : "Course Name"
  CONSTR : TYPE = <STRING>
},
{ ID : "units"
  NAME          : "No OF Units"
  CONSTR : TYPE = <NUMBER>
},
{ ID : "instr"
  NAME : "Initials OF Instructor"
  CONSTR : TYPE = <STRING>
},
```

```
                    { ID : "courses"
                      NAME : "Courses to register"
                      CONSTR : TYPE = < TABLE [ course_no<NUMBER>,
                                        c_NAME<STRING>, units<NUMBER>,
                                                  instructor<STRING> ] >
                    }
DISPLAY METHODS :
          FILLIN DISPLAY {          # This is to fill the Registration form
                    "Registration Form"

                    <FILLIN ITEMS : year, sem>
                    <FILLIN ITEMS : name>
                    <FILLIN ITEMS : roll_no>
                    <FILLIN ITEMS : room_no>
                    <FILLIN ITEMS : courses >

           }
#-------------------------------------------------------------------
CLASS    :        DOAA_INTERFACE
TYPE     :        INTERFACE
INHERITS :  FACULTY_MEMBER_INTERFACE
MENU ITEMS : { ID : "registration"
                  NAME : "Registration related"
                  ACTION : INVOKE_MENU [   "registration/make_reg_notice",
                                           "registration/send_reg_notice",
                                           "registration/send_reg_forms" ]
                },
                { ID : "grades"
                  NAME : "Grades processing"
                  ACTION : INVOKE_MENU [ "grades/prepare_grade_sheets",
                                         "grades/send_grade_sheets_to_depts" ]
```

```
},
{ ID : "degree_award"
  NAME : "degree Awarding"
  ACTION : INVOKE_MENU [ "degree_award/check_for_deg_prereq" ]
},
{ ID : "registration/make_reg_notice"
  NAME : "Prepare Registration Notice"
  ACTION : SEND MESSAGE prepare_reg_notice OF Registration
},
{ ID : "registration/send"
  NAME : "Send Registration Notice"
  ACTION : SEND MESSAGE send_reg_notice_to_depts OF Registration
},
{ ID : "registration/send_reg_forms"
  NAME : "Send Registration forms to DPGCs, DUGCs"
  ACTION : SEND MESSAGE send_reg_forms_to_cdgcs OF
              Registration
},
{ ID : "grades/prepare_grade_sheets"
  NAME : "Prepare Student's Grade Sheets"
  ACTION :SEND MESSAGE prepare_grade_sheets OF Evaluation_proces
},
{ ID : "grades/send_grade_sheets_to_depts"
  NAME : "Send Student's Grade Sheets to their Depts"
  ACTION : SEND MESSAGE students_grade_sheets OF
              Evaluation_process
}
#-------------------------------------------------------------------------
CLASS            : COURSE_COORD_INTERFACE
TYPE             : INTERFACE
MENU ITEMS       :
```

```
                    { ID : "courses"
                      NAME : "Courses Menu"
                      ACTION : INVOKE_MENU ["general/get_registered_students"]
                    },
                    { ID : "courses/give_grades"
                      NAME : "Give grades to registered students"
                      ACTION : SEND MESSAGE give_registered_students OF
                                        Evaluation
                    }
FILLIN ITEMS    : { ID : "student"
                      NAME : "Student Name"
                      CONSTR : TYPE = <STRING>
                    },
                    { ID : "roll_no"
                      NAME : "Roll number"
                      CONSTR : TYPE = <NUMBER>
                    },
                    { ID : "grade"
                      NAME : "Grade"
                      CONSTR : TYPE = <CHAR> && VALUE = [A-F]
                    },
                    { ID : "grade_list"
                      NAME : "List OF Student Grades"
                      CONSTR : TYPE = < TABLE [student<STRING>, roll_no
                                              <NUMBER>, grade_given<CHAR>]>
                    }
DISPLAY METHODS :
          MENU DISPLAY : VERTICAL
#-------------------------------------------------------------------------#
CLASS                    : NOTICE_INTERFACE
TYPE                     : INTERFACE
```

```
FILLIN ITEMS          : { ID : "title"
                          NAME : "Title OF the Notice"
                          CONSTR : TYPE = <STRING>
                          ATTACHED_TO : NOTICE
                        },
                        { ID : "to"
                          NAME : "To"
                          CONSTR : TYPE = <STRING>
                          ATTACHED_TO : NOTICE
                        },
                        { ID : "timestamp"
                          NAME : "Dated"
                          CONSTR : TYPE = <DATE>
                          ATTACHED_TO : NOTICE
                        },
                        { ID : "issuedby"
                          NAME : "Issued by"
                          CONSTR : TYPE = <STRING>
                          ATTACHED_TO : NOTICE
                        },
                        { ID : "text"
                          NAME : "Text OF the Notice"
                          CONSTR : TYPE = <STRING>
                          ATTACHED_TO : NOTICE
                        }

 RELATED WITH     : NOTICE (notice, 1)
 #-------------------------------------------------------------------------

 GENERIC PROCESS :: make_notice
 INPUTS : ISSUEE
```

```
#--------------------------------#

FROM       : ISSUE
TO          : NOTICE_INTERFACE
MSG_ID     : make_notice ()
RESULT     : notice <NOTICE>

FROM       : NOTICE_INTERFACE
TO          : ISSUE
MSG_ID     : replyto_make_notice (notice <NOTICE>)
MSG_TYPE   : reply

#-------------------------------------------------------------------------------#

SUB_PROCESS :: process_the_reg_form
#--------------------------------#

FROM       : DOAA
TO          : SELF
MSG_ID     : extract_details (QUERY = "roll_no" <STRING>,
                    $reg_form!req_for_reg!get_regestered <REGESTRATION_FORM>)
              # extract_details should be taken as a general purpose message
RESULT     : roll_no <NUMBER>

FROM       : DOAA
TO          : HALL_OFFICE
MSG_ID     : check_student_for_hall_dues ($roll_no=extract_details
                              !process_the_reg_form <NUMBER>)
RESULT     : payed <BOOLEAN>

IF ($payed=check_student_for_hall_dues == TRUE) {
```

```
FROM       : DOAA
TO          : ACCOUNTS_SECTION
MSG_ID     : check_student_for_instt_dues ($roll_no=extract_details
                          !process_the_reg_form <NUMBER>)
RESULT     : payed<BOOLEAN>

IF ($payed=check_student_for_instt_dues == TRUE) {

     FROM       : DOAA
     TO          : STUDENT (BACK)
     MSG_ID     : replyto_req_for_reg (remarks = "Registered"
                                          <STRING>)
     MSG_TYPE   : reply

 } ELSE {

     FROM       : DOAA
     TO          : STUDENT (BACK)
     MSG_ID     : replyto_req_for_reg (remarks = "Instt dues not paid"
                                          <STRING>)
     MSG_TYPE   : reply

 }

} ELSE {

    FROM       : DOAA
    TO          : STUDENT (BACK)
    MSG_ID     : replyto_req_for_reg (remarks = "Hall dues not paid" <STRING>)
    MSG_TYPE   : reply
```

```
}
#-------------------------------------------------------------------------#

SUB_PROCESS :: get_regestered
#--------------------------#

FROM          : STUDENT
TO            : SELF
MSG_ID        : check_for_reg_form()
RESULT        : present <BOOLEAN>

IF ($present=check_for_reg_form == TRUE) {

    FROM      : STUDENT
    TO        : SELF
    MSG_ID    : check_if_filled()
    RESULT    : filled  <BOOLEAN>

    IF ($filled=check_if_filled == TRUE) {

        FROM       : STUDENT
        TO         : DOAA(ALL)
        MSG_ID     : req_for_reg ($reg_form=req_for_reg_form!Registration
                                                  <REGESTRATION_FORM>)
        RESULT     : remarks <STRING>
        ACTION     : DO SUB_PROCESS process_the_reg_form

        FROM              : STUDENT
        TO             : STUDENT_INTERFACE (BACK)
        MSG_ID     : replyto_submit_filled_reg_form($remarks=req_for_reg!
```

```
                                                    get_regestered<STRING> )
        MSG_TYPE   : reply
        ACTION     : DO SUB_PROCESS display_remarks($remarks!
                     replyto_submit_filled_reg_form!get_regestered<STRING>)

   }
   ELSE {

        FROM               : STUDENT
        TO              : STUDENT_INTERFACE (BACK)
        MSG_ID     : replyto_submit_filled_reg_form(remarks="Not Filled!"
                                                    <STRING> )
        MSG_TYPE   : reply
        ACTION     : DO SUB_PROCESS display_remarks($remarks!
                     replyto_submit_filled_reg_form!get_regestered<STRING>)
   }
}
ELSE {
      FROM               : STUDENT
      TO              : STUDENT_INTERFACE (BACK)
      MSG_ID     : replyto_submit_filled_reg_form(remarks="No Reg-form present"
                                                  <STRING> )
      MSG_TYPE   : reply
      ACTION     : DO SUB_PROCESS display_remarks($remarks!
                   replyto_submit_filled_reg_form!get_regestered<STRING>)
}
#-------------------------------------------------------------------------#

SUB_PROCESS :: check_eligibility_for_reg
#-----------------------------------#
```

```
FROM       : CONVENER
TO          : SELF
MSG_ID     : check_for_reg_notice ()
RESULT     : present <BOOLEAN>

IF ($present=check_for_reg_notice == TRUE) {

    FROM       : CONVENER
    TO          : SELF
    MSG_ID     : check_elig ()
    RESULT     : eligibile <BOOLEAN>

    IF ($eligible=check_elig == TRUE) {

        FROM       : CONVENER
        TO          : SELF
        MSG_ID     : create_reg_form ()
        RESULT     : reg_form <REGESTRATION_FORM>

        FROM       : CONVENER
        TO          : STUDENT (BACK)
        MSG_ID     : replyto_req_for_reg_form ( $reg_form=create_reg_form
         <REGESTRATION_FORM> ,remarks = "Eligible for Registration" <STRING>)
        MSG_TYPE   : reply
        ACTION     : DO SUB_PROCESS store_reg_form

    } ELSE {  # eligible is FALSE

        FROM       : CONVENER
        TO          : STUDENT (BACK)
        MSG_ID     : replyto_req_for_reg_form ( remarks =
```

```
                                        "Not eligible for registration" <STRING>)
                        # reg_form will be NULL

        MSG_TYPE   : reply
        ACTION     : DO SUB_PROCESS store_reg_form

    }

} ELSE { # Registration notice has not been received from DOAA
          # means it is not yet time for registration

    FROM       : CONVENER
    TO         : STUDENT (BACK)
    MSG_ID     : replyto_req_for_reg_form (remarks =
                                        "Not Yet Time for registration" <STRING>)
    MSG_TYPE   : reply
    ACTION     : DO SUB_PROCESS store_reg_form

}

#-------------------------------------------------------------------------------#

PROCESS :: Registration
#----------------------#

FROM       : DOAA_INTERFACE
TO          : SELF
MSG_ID     : prepare_reg_notice ()
ACTION     : DO GENERIC SUB_PROCESS make_notice
RESULT     : notice <NOTICE>
```

```
FROM       : DOAA_INTERFACE
TO          : DOAA (REL)
MSG_ID     : send_reg_notice_to_depts ($notice=prepare_reg_notice
                                              !Registration<NOTICE>)
PRE_COND   : RECEIVED RESULTOF prepare_reg_notice
ACTION     : SEND MESSAGE notice_for_registration

FROM       : DOAA
TO          : CONVENER (ALL)
MSG_ID     : notice_for_registration ($notice!send_reg_notice_to_depts
                                              !Registration<NOTICE>)
ACTION     : DO SUB_PROCESS store_notice

FROM       : STUDENT_INTERFACE
TO          : STUDENT (REL)
MSG_ID     : get_reg_form ()
RESULT           : remarks <STRING>
ACTION     : SEND MESSAGE req_for_reg_form

FROM       : STUDENT
TO          : CONVENER ( REL )
MSG_ID     : req_for_reg_form ()
RESULT     : res_val {reg_form <REGESTRATION_FORM>, remarks <STRING>}
ACTION     : DO SUB_PROCESS check_eligibility_for_reg

FROM       : STUDENT
TO          : STUDENT_INTERFACE ( BACK )
MSG_ID     : replyto_get_reg_form($remarks=req_for_reg_form!Registration
                                                        <STRING>)
MSG_TYPE   : reply
ACTION     : DO SUB_PROCESS display_remarks($remarks!replyto_get_reg_form
```

```
                                                    !Registration<STRING>)

# sending message to abstract class 'CONVENER' means sending to one of the
# derived classes

FROM       : STUDENT_INTERFACE
TO         : STUDENT(REL)
MSG_ID     : fill_reg_form ()
POST_COND  : RECEIVED RESULTOF req_for_reg_form OF Registration

FROM       : STUDENT_INTERFACE
TO         : STUDENT(REL)
MSG_ID     : submit_filled_reg_form()
ACTION     : DO SUB_PROCESS get_regestered
RESULT     : remarks <STRING>

#-------------------------------------------------------------------------#
```

Appendix B

Notation in BNF

In this appendix a BNF grammar for the specification that is used by the interpreter is given.

```
systemspec          :  classlist processlist genericmsgs
                    ;

/*************************************************************************

                    CLASS AND INTERFACE CLASS

*************************************************************************/
classlist           :  classdecl
                    |  classlist   classdecl
                    ;
classdecl           :  CLASS ':' classname  option_list
                    ;
option_list         :  /* nothing */
                    |  option_list TYPE ':' clstype
                    |  option_list ATTR ':' attr_list
                    |  option_list IFACE ':' ID
                    |  option_list KEY ':' ID
```

```
                | option_list GEN OF ':' classname_list
                | option_list INHERITS ':' classname_list
                | option_list REL WITH ':' rel_list
                | option_list AGGR OF ':' agglist
                | option_list PART OF ':' agglist
                | option_list MENU ITEMS ':' menu_items
                | option_list FILLIN ITEMS ':' fillin_items
                | option_list displ_methods
                ;
clstype         : ID
                | clstype ',' ID
                ;
classname       : ID
                ;
classname_list  : ID
                | classname_list ',' ID
                ;
attr_list       : /* nothing */
                | attr
                | attr_list ',' attr
                ;
attr            : ID LESSER attrtype GREATER
                ;
attrtype        : sim
                | c_table
                | c_set
                | c_list
                ;
sim             : ID
                ;
c_table         : TABL '[' attr_list ']'
```

```
                ;
c_set           : '{'  attr_list '}'
                ;
c_list          : '['  attrtype ']'
                ;
agglist         : aggof
                | agglist ',' aggof
                ;
aggof           : ID '(' multiplicity ')'
                ;
rel_list        : relation
                | rel_list ',' relation
                ;
relation        : ID '(' reltype ',' multiplicity ')'
                ;
multiplicity    : lmcity '-' umcity
                | umcity
                ;
reltype         : ID
                ;
umcity          : 'N'
                | CARDINALITY
                ;
lmcity          : 'M'
                | CARDINALITY
                ;
menu_items      : menu_item
                | menu_items ',' menu_item
                ;
menu_item       : '{' id name menu_action descript '}'
                ;
```

```
id                  :  IDENT ':' STRNG
                    ;
name                :  NAME ':' STRNG
                    ;
descript            :  /* nothing */
                    |  DESCR ':' STRNG
                    ;
menu_action         :  ACTION ':' maction_list
                    ;
maction_list        :  maction
                    |  maction_list ';' maction
                    ;
maction             :  SEND MSG mesg_id
                    |  SEND MSG mesg_id  OF proc_id
                    |  DO SUB_PROCESS sub_proc_id
                    |  INVK_MENU '[' mid_list ']'
                    |  SEL MENU ITEM STRNG
                    |  SEL FILLIN_ITEM STRNG
                    ;
mid_list            :  STRNG
                    |  mid_list ',' STRNG
                    ;
fillin_items        :  fillin_item
                    |  fillin_items ',' fillin_item
                    ;
fillin_item         :  '{' id name constr faction attached_to '}'
                    ;
faction             :  /* nothing */
                    |  menu_action
                    ;
attached_to         :  /* nothing */
```

```
                | ATTACHED_TO ':' ID
                ;
constr          : CONSTR ':' typecon
                | CONSTR ':' typecon AND valcon
                ;
typecon         : TYPE EQL LESSER attrtype GREATER
                ;
valcon          : VALUE EQL '[' valrange ']'
                ;
valrange        : lval
                | lval '-' rval
                ;
lval            : CARDINALITY
                | ID
                ;
rval            : CARDINALITY
                | ID
                ;
displ_methods   : DISPLAY METHODS ':' method
                ;
method          : disp_menu
                | disp_fill
                ;
disp_menu       : MENU DISPLAY ':' mdtype
                ;
mdtype          : HRZ
                | VRT
                ;
disp_fill       : FILLIN DISPLAY '{' STRNG fillin_list '}'
                ;
fillin_list     : fillin
```

```
                | fillin_list fillin
                ;
fillin          : LESSER FILLIN ITEMS ':' finlist GREATER
                ;
finlist         : ID
                | finlist ',' ID
                ;
/*************************************************************************

                PROCESS, SUBPROCESS AND GENERIC SUBPROCESS

*************************************************************************/
processlist     : gen_proclist sub_proclist proclist
                ;
gen_proclist    : /* nothing */
                | gen_process
                | gen_proclist gen_process
                ;
gen_process     : GENERIC PROCESS SEP pid
                | GENERIC PROCESS SEP pid INPUTS ':' varlist
                | GENERIC PROCESS SEP pid INPUTS ':' varlist mesg_sequence
                ;
varlist         : ID
                | varlist ',' ID
                ;
sub_proclist    : /* nothing */
                | sub_process
                | sub_proclist sub_process
                ;
sub_process     : SUB_PROCESS SEP pid
                | SUB_PROCESS SEP pid  mesg_sequence
```

```
                ;
pid             : ID
                ;
proclist        : process
                | proclist process
                ;
process         : PROCESS SEP pid
                | PROCESS SEP pid  mesg_sequence
                ;
mesg_sequence   : sequence
                | mesg_sequence sequence
                ;
sequence        : msglist
                | if_block
                | loop_block
                ;
if_block        : IF '(' conditionList ')' '{' mesg_sequence '}' else_part
                ;
else_part       : /* nothing */
                | ELSE '{' mesg_sequence '}'
                ;
loop_block      : loopcond '{' mesg_sequence '}'
                ;
loopcond        : FOREACH field '(' listOrtable ')'
                | WHILE '(' conditionList ')'
                ;
field           : ID
                ;
listOrtable     : var
                ;
msglist         : msgdecl
```

```
                | msglist msgdecl
                ;
msgdecl         : sender recvr msg_id optlist
                ;
sender          : FROM ':' ID constraint
                ;
recvr           : TO ':' ID constraint
                ;
constraint      : /* nothing */
                | '(' cond ')'
                ;
cond            : lhs operator rhs
                | reference
                ;
lhs             : var
                | ID SEP ID
                ;
rhs             : var
                | ID SEP ID
                ;
reference       : var
                | DOLLAR FROM '{' mesg_id procid '}'
                ;
operator        : EQUAL
                | NOT_EQUAL
                | LESSER
                | LESSER_EQUAL
                | GREATER
                | GREATER_EQUAL
                ;
msg_id          : MSG_ID ':' message
```

```
                ;
message         : mesg_id '(' ')'
                | mesg_id '(' paramlist ')'
                ;
mesg_id         : ID
                ;
paramlist       : param
                | paramlist ',' param
                ;
param           : var
                | var LESSER type GREATER
                | var EQL STRNG LESSER type GREATER
                | query
                ;
var             : ID
                | DOLLAR source
                | PERCENT ID
                ;
source          : arg_or_res mesgid procid
                | nest_level '~' ID
                ;
nest_level      : CARDINALITY
                ;
procid          : /* optional */
                | NOT proc_id
                ;
mesgid          : /* optional */
                | sign mesg_id
                ;
sign            : NOT
                | EQL
```

```
                ;
arg_or_res      : ID
                ;
query           : QUERY EQL '"' qlist '"' LESSER type GREATER
                ;
qlist           : ID
                | qlist ',' ID
                ;
type            : ID
                | m_list
                | m_table
                ;
m_list          : '[' type ']'
                ;
m_table         : TABL '[' paramlist ']'
                ;
optlist         : /* nothing */
                | optlist msg_cond
                | optlist msgtype
                | optlist result
                | optlist action
                | optlist precond
                | optlist postcond
                ;
msgtype         : MSGTYPE ':' ID
                ;
msg_cond        : MSG_COND ':' mcond
                ;
mcond           : constraint
                ;
precond         : PRE_COND ':' conditionList
```

```
                ;
conditionList   : condition
                | conditionList l_operator condition
                ;
l_operator      : AND
                | OR
                | NOT
                ;
condition       : RECVD mesg_id
                | RECVD mesg_id OF proc_id
                | RECVD RESULT_OF mesg_id
                | RECVD RESULT_OF mesg_id OF proc_id
                | DONE SUB_PROCESS proc_id
                | cond
                ;
proc_id         : ID
                ;
postcond        : POST_COND ':' conditionList
                ;
result          : RESULT ':' res
                ;
res             : attr
                | set_of_values
                ;
set_of_values   : ID '{' attr_list '}'
                ;
action          : ACTION ':' action_list
                ;
action_list     : actual_action
                | action_list ',' actual_action
                ;
```

```
actual_action     :  DO SUB_PROCESS sub_proc_id sub_proc_args
                  |  DO GENERIC SUB_PROCESS sub_proc_id sub_proc_args
                     gen_values
                  |  RETREIVE res
                  |  SEND MSG mesg_id to_recvr OF proc_id sub_proc_args
                  |  SEND MSG mesg_id to_recvr sub_proc_args
                  |  SEND GENERIC MSG mesg_id to_recvr OF proc_id inputs
                  |  SEL MENU ITEM item
                  ;
item              : ID
                  ;
to_recvr          :  /* nothing */
                  |  TO ID
                  ;
gen_values        :  /* nothing */
                  |  '{' value_list '}'
                  ;
value_list        :  value
                  |  value_list ',' value
                  ;
value             :  ID ':' replacing_val
                  ;
replacing_val     :  ID
                  |  DOLLAR ID
                  ;
inputs            :  '{' input_list '}'
                  ;
input_list        :  var_value_pair
                  |  input_list ',' var_value_pair
var_value_pair          :  TO ':' ID
                  |  FROM ':' ID
```

```
                ;
sub_proc_args   :  /* nothing */
                |  '(' paramlist ')'
                ;
sub_proc_id     :  ID
                ;

/*************************************************************************

                        GENERIC MESSAGES

*************************************************************************/
genericmsgs     :  /* nothing */
                |  gen_messages
                ;
gen_messages    :  GENERIC MSG SEP gmsglist
                ;
gmsglist        :  gmsg_decl
                |  gmsglist gmsg_decl
                ;
gmsg_decl       :  gsender grecvr msg_id optlist
                ;
gsender         :  FROM ':' gen_clsname constraint
                ;
grecvr          :  TO ':' gen_clsname constraint
                ;
gen_clsname     :  '*'
                |  ID
                ;
```

Bibliography

[Boo94] Grady Booch. *Object Oriented Design and Applications.* Benjamin Cummins, 1994.

[Hol] A Holub. *Compiler Design in C.* Prentice Hall.

[R+91] James Rumbaugh et al. *Object Oriented Modelling and Design.* Prentice Hall, 1991.

[Raf96] D Rafee. Translating message centric oo specification to c++. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, March 1996.

[Sar96] B Ramanjaneya Sarma. A message based specification system for object oriented software construction. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, February 1996.

[Shr97] R K Shrivasthava. An approach to testing message centric oo specifications. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, April 1997.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2 edition, 1991.

[Vij96] M Vijayanand. Translating message centric oo specification to rdbms. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, March 1996.